libuv 中文教程

wizardforcel

Published with GitBook



目錄

libuv中文教程

翻译自《An Introduction to libuv》.
会持续关注原教程并更新中文版，本教程基于libuv的v1.3.0。

在线阅读

使用gitbook制作，可以在线阅读。

目录

（文章源文件均在 `source/` 文件夹下）

翻译人员

- byronhe
- littleneko
- luohaha

辅助阅读

1. libuv官方文档—由于教程有些知识点讲解得不够深入，需要我们自行阅读官方文档，来加强理解。
2. 教程的完整代码—教程中展示的代码并不完整，对于一些复杂的程序，需要阅读完整的实例代码。

说明

在翻译的过程中，对于一些个人觉得可能不是那么容易理解的知识点，我都会附上自己收集的说明资料的链接，以方便学习。由于个人的英文水平有限，如果大家发现翻译出错或者不合适的地方，欢迎PR（修改master分支下，source文件夹下的md文件即可）。

Introduction

本书由很多的libuv教程组成，libuv是一个高性能的，事件驱动的I/O库，并且提供了跨平台（如windows, linux）的API。

本书会涵盖libuv的主要部分，但是不会详细地讲解每一个函数和数据结构。官方文档中可以查阅到完整的内容。

本书依然在不断完善中，所以有些章节会不完整，但我希望你能喜欢它。

Who this book is for

如果你正在读此书，你或许是：

> 1. 系统程序员，会编写一些底层的程序，例如守护进程或者网络服务器／客户端。你也许发现了event-loop很适合于你的应用场景，然后你决定使用libuv。
> 2. 一个node.js的模块开发人员，决定使用C/C++封装系统平台某些同步或者异步API，并将其暴露给Javascript。你可以在node.js中只使用libuv。但你也需要参考其他资源，因为本书并没有包括v8/node.js相关的内容。

本书假设你对c语言有一定的了解。

Background

node.js最初开始于2009年，是一个可以让Javascript代码离开浏览器的执行环境也可以执行的项目。 node.js使用了Google的V8解析引擎和Marc Lehmann的libev。Node.js将事件驱动的I/O模型与适合该模型的编程语言(Javascript)融合在了一起。随着node.js的日益流行，node.js需要同时支持windows, 但是libev只能在Unix环境下运行。Windows 平台上与kqueue(FreeBSD)或者(e)poll(Linux)等内核事件通知相应的机制是IOCP。libuv提供了一个跨平台的抽象，由平台决定使用libev或IOCP。在node-v0.9.0版本中，libuv移除了libev的内容。

随着libuv的日益成熟，它成为了拥有卓越性能的系统编程库。除了node.js以外，包括Mozilla的Rust编程语言，和许多的语言都开始使用libuv。

本书基于libuv的v1.3.0。

Code

本书中的实例代码都可以在Github上找到。

Basics of libuv

libuv强制使用异步的，事件驱动的编程风格。它的核心工作是提供一个event-loop，还有基于I/O和其它事件通知的回调函数。libuv还提供了一些核心工具，例如定时器，非阻塞的网络支持，异步文件系统访问，子进程等。

Event loops

在事件驱动编程中，程序会关注每一个事件，并且对每一个事件的发生做出反应。libuv会负责将来自操作系统的事件收集起来，或者监视其他来源的事件。这样，用户就可以注册回调函数，回调函数会在事件发生的时候被调用。event-loop会一直保持运行状态。用伪代码描述如下：

```
while there are still events to process:
    e = get the next event
    if there is a callback associated with e:
        call the callback
```

举几个事件的例子：

> - 准备好被写入的文件。
> - 包含准备被读取的数据的socket。
> - 超时的定时器。

event-loop最终会被 `uv_run()` 启动－当使用libuv时，最后都会调用的函数。

系统编程中最经常处理的一般是输入和输出，而不是一大堆的数据处理。问题在于传统的输入／输出函数(例如 `read` ， `fprintf`)都是阻塞式的。实际上，向文件写入数据，从网络读取数据所花的时间，对比cpu的处理速度差得太多。任务没有完成，函数是不会返回的，所以你的程序在这段时间内什么也做不了。对于需要高性能的的程序来说，这是一个主要的障碍。

其中一个标准的解决方案是使用多线程。每一个阻塞的I/O操作都会被分配到各个线程中（或者是使用线程池）。当某个线程一旦阻塞，处理器就可以调度处理其他需要cpu资源的线程。

但是libuv使用了另外一个解决方案，那就是异步，非阻塞。大多数的现代操作系统提供了基于事件通知的子系统。例如，一个正常的socket上的 `read` 调用会发生阻塞，直到发送方把信息发送过来。但是，实际上程序可以请求操作系统监视socket事件的到来，并将这个事件通知放到事件队列中。这样，程序就可以很简单地检查事件是否到来（可能此时正在使用cpu做数值处理的运算），并及时地获取数据。说libuv是异步的，是因为程序可以在一头表达对某一事件的兴趣，并在另一头获取到数据（对于时间或是空间来说）。它是非阻塞是因为应用程序无需在请求数据后

等待，可以自由地做其他的事。libuv的事件循环方式很好地与该模型匹配, 因为操作系统事件可以视为另外一种libuv事件. 非阻塞方式可以保证在其他事件到来时被尽快处理（当然还要考虑硬件的能力）。

Note

> 我们不需要关心I/O在后台是如何工作的，但是由于我们的计算机硬件的工作方式，线程是处理器最基本的执行单元，libuv和操作系统通常会运行后台/工作者线程, 或者采用非阻塞方式来轮流执行任务。

Bert Belder，一个libuv的核心开发者，通过一个短视频向我们解释了libuv的架构和它的后台工作方式。如果你之前没有接触过类似libuv，libev，这个视频会非常有用。视频的网址是https://youtu.be/nGn60vDSxQ4 。

包含了libuv的event-loop的更多详细信息的文档。

HELLO WORLD

让我们开始写第一个libuv程序吧！它什么都没做，只是开启了一个loop，然后很快地退出了。

helloworld/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>

int main() {
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    uv_loop_init(loop);

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    uv_loop_close(loop);
    free(loop);
    return 0;
}
```

这个程序会很快就退出了，因为没有可以很处理的事件。我们可以使用各种API函数来告诉event-loop我们要监视的事件。

从libuv的1.0版本开始，用户就可以在使用 `uv_loop_init` 初始化loop之前，给其分配相应的内存。这就允许你植入自定义的内存管理方法。记住要使用 `uv_loop_close(uv_loop_t *)` 关闭loop，然后再回收内存空间。在例子中，程序退出的时候会关闭loop，系统也会自动回收内存。对于长时间运行的程序来说，合理释放内存很重要。

Default loop

可以使用 `uv_default_loop` 获取libuv提供的默认loop。如果你只需要一个loop的话，可以使用这个。

Note

> nodejs中使用了默认的loop作为自己的主loop。如果你在编写nodejs的绑定，你应该注意一下。

Error handling

初始化函数或者是同步执行的函数，会在执行失败后返回代表错误的负数。但是对于异步执行的函数，会在执行失败的时候，给它们的回调函数传递一个状态参数。错误信息被定义为 `UV_E` 常量。

你可以使用 `uv_strerror(int)` 和 `uv_err_name(int)` 分别获取 `const char *` 格式的错误信息和错误名字。

I/O函数的回调函数（例如文件和socket等）会被传递一个 `nread` 参数。如果 `nread` 小于0，就代表出现了错误（当然，UV_EOF是读取到文件末端的错误，你要特殊处理）。

Handles and Requests

libuv的工作建立在用户表达对特定事件的兴趣。这通常通过创造对应I/O设备，定时器，进程等的handle来实现。handle是不透明的数据结构，其中对应的类型 `uv_TYPE_t` 中的type指定了handle的使用目的。

libuv watchers

```
/* Handle types. */
typedef struct uv_loop_s uv_loop_t;
typedef struct uv_handle_s uv_handle_t;
typedef struct uv_stream_s uv_stream_t;
typedef struct uv_tcp_s uv_tcp_t;
typedef struct uv_udp_s uv_udp_t;
typedef struct uv_pipe_s uv_pipe_t;
typedef struct uv_tty_s uv_tty_t;
typedef struct uv_poll_s uv_poll_t;
typedef struct uv_timer_s uv_timer_t;
typedef struct uv_prepare_s uv_prepare_t;
typedef struct uv_check_s uv_check_t;
typedef struct uv_idle_s uv_idle_t;
typedef struct uv_async_s uv_async_t;
typedef struct uv_process_s uv_process_t;
typedef struct uv_fs_event_s uv_fs_event_t;
typedef struct uv_fs_poll_s uv_fs_poll_t;
typedef struct uv_signal_s uv_signal_t;

/* Request types. */
typedef struct uv_req_s uv_req_t;
typedef struct uv_getaddrinfo_s uv_getaddrinfo_t;
typedef struct uv_getnameinfo_s uv_getnameinfo_t;
typedef struct uv_shutdown_s uv_shutdown_t;
typedef struct uv_write_s uv_write_t;
typedef struct uv_connect_s uv_connect_t;
typedef struct uv_udp_send_s uv_udp_send_t;
typedef struct uv_fs_s uv_fs_t;
typedef struct uv_work_s uv_work_t;

/* None of the above. */
typedef struct uv_cpu_info_s uv_cpu_info_t;
typedef struct uv_interface_address_s uv_interface_address_t;
typedef struct uv_dirent_s uv_dirent_t;
```

handle代表了持久性对象。在异步的操作中，相应的handle上有许多与之关联的request。request是短暂性对象（通常只维持在一个回调函数的时间），通常对映着handle上的一个I/O操作。request用来在初始函数和回调函数之间，传递上下文。例如uv_udp_t代表了一个udp的socket，然而，对于每一个向socket的写入的完成后，都会向回调函数传递一个 `uv_udp_send_t` 。

handle可以通过下面的函数设置：

```
uv_TYPE_init(uv_loop_t *, uv_TYPE_t *)
```

回调函数是libuv所关注的事件发生后，所调用的函数。应用程序的特定逻辑会在回调函数中实现。例如，一个IO监视器的回调函数会接收到从文件读取到的数据，一个定时器的回调函数会在超时后被触发等等。

Idling

下面有一个使用空转handle的例子。回调函数在每一个循环中都会被调用。在Utilities这部分会讲到一些空转handle的使用场景。现在让我们使用一个空转监视器，然后来观察它的生命周期，接着看 `uv_run` 调用是否会造成阻塞。当达到事先规定好的计数后，空转监视器会退出。因为 `uv_run` 已经找不到活着的事件监视器了，所以 `uv_run()` 也退出。

idle-basic/main.c

```
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

void wait_for_a_while(uv_idle_t* handle) {
    counter++;

    if (counter >= 10e6)
        uv_idle_stop(handle);
}

int main() {
    uv_idle_t idler;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, wait_for_a_while);

    printf("Idling...\n");
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    uv_loop_close(uv_default_loop());
    return 0;
}
```

Storing context

在基于回调函数的编程风格中，你可能会需要在调用处和回调函数之间，传递一些上下文等特定的应用信息。所有的handle和request都有一个 `data` 域，可以用来存储信息并传递。这是一个c语言库中很常见的模式。即使是 `uv_loop_t` 也有一个相似的 `data` 域。

Filesystem

简单的文件读写是通过 `uv_fs_*` 函数族和与之相关的 `uv_fs_t` 结构体完成的.

note

> libuv 提供的文件操作和 socket operations 并不相同. 套接字操作使用了操作系统本身提供了非阻塞操作, 而文件操作内部使用了阻塞函数, 但是 libuv 是在线程池中调用这些函数, 并在应用程序需要交互时通知在事件循环中注册的监视器.

所有的文件操作函数都有两种形式 - 同步 synchronous 和 异步 asynchronous.

同步 synchronous 形式如果没有指定回调函数则会被自动调用(并阻塞的), 函数的返回值是 `libuv error code` . 但以上通常只对同步调用有意义.

而异步 asynchronous 形式则会在传入回调函数时被调用, 并且返回 0.

Reading/Writing files

文件描述符可以采用如下方式获得:

```
int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path, int
```

参数 `flags` 与 `mode` 和标准的 Unix flags 相同. libuv 会小心地处理 Windows 环境下的相关标志位(flags)的转换, 所以编写跨平台程序时你不用担心不同平台上文件打开的标志位不同。

关闭文件描述符可以使用:

```
int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_
```

文件系统的回调函数有如下的形式：

```
void callback(uv_fs_t* req);
```

让我们看一下一个简单的 `cat` 命令的实现。我们通过注册一个当文件被打开时被调用的回调函数来开始：

uvcat/main.c - opening a file

```
    // The request passed to the callback is the same as the one th
    // function was passed.
    assert(req == &open_req);
    if (req->result >= 0) {
        iov = uv_buf_init(buffer, sizeof(buffer));
        uv_fs_read(uv_default_loop(), &read_req, req->result,
                   &iov, 1, -1, on_read);
    }
    else {
        fprintf(stderr, "error opening file: %s\n", uv_strerror((in
    }
}
```

`uv_fs_t` 的 `result` 域保存了 `uv_fs_open` 回调函数打开的文件描述符。如果文件被正确地打开，我们可以开始读取了：

```
void on_read(uv_fs_t *req) {
    if (req->result < 0) {
        fprintf(stderr, "Read error: %s\n", uv_strerror(req->result
    }
    else if (req->result == 0) {
        uv_fs_t close_req;
        // synchronous
        uv_fs_close(uv_default_loop(), &close_req, open_req.result,
    }
    else if (req->result > 0) {
        iov.len = req->result;
        uv_fs_write(uv_default_loop(), &write_req, 1, &iov, 1, -1,
    }
}
```

在调用读取函数的时候，你必须传递一个已经初始化的缓冲区，在 `on_read()` 被触发后，缓冲区被被写入数据。 `uv_fs_*` 系列的函数是和POSIX的函数对应的，所以当读到文件的末尾时(EOF)，result返回0。在使用streams或者pipe的情况下，使用的是libuv自定义的 `UV_EOF` 。

现在你看到类似的异步编程的模式。但是 `uv_fs_close()` 是同步的，一般来说，一次性的，开始的或者关闭的部分，都是同步的，因为我们一般关心的主要是任务和多路I/O的快速I/O。所以在这些对性能微不足道的地方，都是使用同步的，这样代码还会简单一些。

文件系统的写入使用 `uv_fs_write()` ，当写入完成时会触发回调函数，在这个例子中回调函数会触发下一次的读取。

uvcat/main.c - write callback

```
void on_write(uv_fs_t *req) {
    if (req->result < 0) {
        fprintf(stderr, "Write error: %s\n", uv_strerror((int)req->
    }
    else {
        uv_fs_read(uv_default_loop(), &read_req, open_req.result, &
    }
}
```

Warning

> 由于文件系统和磁盘的调度策略，写入成功的数据不一定就存在磁盘上。

我们开始在main中推动多米诺骨牌：

uvcat/main.c

```
int main(int argc, char **argv) {
    uv_fs_open(uv_default_loop(), &open_req, argv[1], O_RDONLY, 0,
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    uv_fs_req_cleanup(&open_req);
    uv_fs_req_cleanup(&read_req);
    uv_fs_req_cleanup(&write_req);
    return 0;
}
```

Warning

> 函数uv_fs_req_cleanup()在文件系统操作结束后必须要被调用，用来回收在读写中分配的内存。

Filesystem operations

所有像 `unlink` , `rmdir` , `stat` 这样的标准文件操作都是支持异步的，并且使用方法和上述类似。下面的各个函数的使用方法和read/write/open类似，在 `uv_fs_t.result` 中保存返回值.所有的函数如下所示：
（译者注：返回的result值，<0表示出错，其他值表示成功。但>=0的值在不同的函数中表示的意义不一样，比如在 `uv_fs_read` 或者 `uv_fs_write` 中，它代表读取或写入的数据总量，但在 `uv_fs_open` 中表示打开的文件描述符.）

```
UV_EXTERN int uv_fs_close(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
```

```
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_open(uv_loop_t* loop,
                         uv_fs_t* req,
                         const char* path,
                         int flags,
                         int mode,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_read(uv_loop_t* loop,
                         uv_fs_t* req,
                         uv_file file,
                         const uv_buf_t bufs[],
                         unsigned int nbufs,
                         int64_t offset,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_unlink(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_write(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          const uv_buf_t bufs[],
                          unsigned int nbufs,
                          int64_t offset,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_mkdir(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          int mode,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_mkdtemp(uv_loop_t* loop,
                            uv_fs_t* req,
                            const char* tpl,
                            uv_fs_cb cb);
UV_EXTERN int uv_fs_rmdir(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_scandir(uv_loop_t* loop,
                            uv_fs_t* req,
                            const char* path,
                            int flags,
                            uv_fs_cb cb);
UV_EXTERN int uv_fs_scandir_next(uv_fs_t* req,
                                 uv_dirent_t* ent);
UV_EXTERN int uv_fs_stat(uv_loop_t* loop,
                         uv_fs_t* req,
                         const char* path,
                         uv_fs_cb cb);
UV_EXTERN int uv_fs_fstat(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          uv_fs_cb cb);
```

```
UV_EXTERN int uv_fs_rename(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           const char* new_path,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_fsync(uv_loop_t* loop,
                          uv_fs_t* req,
                          uv_file file,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_fdatasync(uv_loop_t* loop,
                              uv_fs_t* req,
                              uv_file file,
                              uv_fs_cb cb);
UV_EXTERN int uv_fs_ftruncate(uv_loop_t* loop,
                              uv_fs_t* req,
                              uv_file file,
                              int64_t offset,
                              uv_fs_cb cb);
UV_EXTERN int uv_fs_sendfile(uv_loop_t* loop,
                             uv_fs_t* req,
                             uv_file out_fd,
                             uv_file in_fd,
                             int64_t in_offset,
                             size_t length,
                             uv_fs_cb cb);
UV_EXTERN int uv_fs_access(uv_loop_t* loop,
                           uv_fs_t* req,
                           const char* path,
                           int mode,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_chmod(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          int mode,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_utime(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          double atime,
                          double mtime,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_futime(uv_loop_t* loop,
                           uv_fs_t* req,
                           uv_file file,
                           double atime,
                           double mtime,
                           uv_fs_cb cb);
UV_EXTERN int uv_fs_lstat(uv_loop_t* loop,
                          uv_fs_t* req,
                          const char* path,
                          uv_fs_cb cb);
UV_EXTERN int uv_fs_link(uv_loop_t* loop,
                         uv_fs_t* req,
```

```
                         const char* path,
                         const char* new_path,
                         uv_fs_cb cb);
```

Buffers and Streams

在libuv中，最基础的I/O操作是流stream(`uv_stream_t`)。TCP套接字，UDP套接字，管道对于文件I/O和IPC来说，都可以看成是流stream(`uv_stream_t`)的子类.上面提到的各个流的子类都有各自的初始化函数，然后可以使用下面的函数操作:

```
int uv_read_start(uv_stream_t*, uv_alloc_cb alloc_cb, uv_read_cb re
int uv_read_stop(uv_stream_t*);
int uv_write(uv_write_t* req, uv_stream_t* handle,
                const uv_buf_t bufs[], unsigned int nbufs, uv_writ
```

可以看出，流操作要比上述的文件操作要简单一些，而且当 `uv_read_start()` 一旦被调用，libuv会保持从流中持续地读取数据，直到 `uv_read_stop()` 被调用。数据的离散单元是buffer- `uv_buffer_t` 。它包含了指向数据的开始地址的指针(`uv_buf_t.base`)和buffer的长度(`uv_buf_t.len`)这两个信息。 `uv_buf_t` 很轻量级，使用值传递。我们需要管理的只是实际的数据，即程序必须自己分配和回收内存。

.. ERROR::

```
THIS PROGRAM DOES NOT ALWAYS WORK, NEED SOMETHING BETTER**
```

为了更好地演示流stream，我们将会使用 `uv_pipe_t` 。它可以将本地文件转换为流（stream）的形态。接下来的这个是使用libuv实现的，一个简单的T型工具（如果不是很了解，请看维基百科)）。所有的操作都是异步的，这也正是事件驱动I/O的威力所在。两个输出操作不会相互阻塞，但是我们也必须要注意，确保一块缓冲区不会在还没有写入之前，就提前被回收了。

这个程序执行命令如下

```
./uvtee <output_file>
```

在使用pipe打开文件时，libuv会默认地以可读和可写的方式打开文件。

uvtee/main.c - read on pipes

```
int main(int argc, char **argv) {
    loop = uv_default_loop();

    uv_pipe_init(loop, &stdin_pipe, 0);
    uv_pipe_open(&stdin_pipe, 0);

    uv_pipe_init(loop, &stdout_pipe, 0);
    uv_pipe_open(&stdout_pipe, 1);

    uv_fs_t file_req;
    int fd = uv_fs_open(loop, &file_req, argv[1], O_CREAT | O_RDWR,
    uv_pipe_init(loop, &file_pipe, 0);
    uv_pipe_open(&file_pipe, fd);

    uv_read_start((uv_stream_t*)&stdin_pipe, alloc_buffer, read_std

    uv_run(loop, UV_RUN_DEFAULT);
    return 0;
}
```

当需要使用IPC的命名管道的时候（无名管道是*Unix*最初的*IPC*形式，但是由于无名管道的局限性，后来出现了有名管道*FIFO*，这种管道由于可以在文件系统中创建一个名字，所以可以被没有亲缘关系的进程访问），`uv_pipe_init()` 的第三个参数应该被设置为1。这部分会在Process进程的这一章节说明。`uv_pipe_open()` 函数把管道和文件描述符关联起来，在上面的代码中表示把管道 `stdin_pipe` 和标准输入关联起来（译者注：`0` 代表标准输入，`1` 代表标准输出，`2` 代表标准错误输出）。

当调用 `uv_read_start()` 后，我们开始监听 `stdin` ，当需要新的缓冲区来存储数据时，调用alloc_buffer，在函数 `read_stdin()` 中可以定义缓冲区中的数据处理操作。

uvtee/main.c - reading buffers

```
void alloc_buffer(uv_handle_t *handle, size_t suggested_size, uv_bu
    *buf = uv_buf_init((char*) malloc(suggested_size), suggested_si
}

void read_stdin(uv_stream_t *stream, ssize_t nread, const uv_buf_t
    if (nread < 0){
        if (nread == UV_EOF){
            // end of file
            uv_close((uv_handle_t *)&stdin_pipe, NULL);
            uv_close((uv_handle_t *)&stdout_pipe, NULL);
            uv_close((uv_handle_t *)&file_pipe, NULL);
        }
    } else if (nread > 0) {
        write_data((uv_stream_t *)&stdout_pipe, nread, *buf, on_std
        write_data((uv_stream_t *)&file_pipe, nread, *buf, on_file_
    }

    if (buf->base)
        free(buf->base);
}
```

标准的 `malloc` 是非常高效的方法，但是你依然可以使用其它的内存分配的策略。比如，nodejs使用自己的内存分配方法（ `Smalloc` ），它将buffer用v8的对象关联起来，具体的可以查看nodejs的官方文档。

当回调函数 `read_stdin()` 的nread参数小于0时，表示错误发生了。其中一种可能的错误是EOF(读到文件的尾部)，这时我们可以使用函数 `uv_close()` 关闭流了。除此之外，当nread大于0时，nread代表我们可以向输出流中写入的字节数目。最后注意，缓冲区要由我们手动回收。

当分配函数 `alloc_buf()` 返回一个长度为0的缓冲区时，代表它分配内存失败。在这种情况下，读取的回调函数会被错误 `UV_ENOBUFS` 唤醒。libuv同时也会继续尝试从流中读取数据，所以如果你想要停止的话，必须明确地调用 `uv_close()` .

当nread为0时，代表已经没有可读的了，大多数的程序会自动忽略这个。

uvtee/main.c - Write to pipe

```
typedef struct {
    uv_write_t req;
    uv_buf_t buf;
} write_req_t;

void free_write_req(uv_write_t *req) {
    write_req_t *wr = (write_req_t*) req;
    free(wr->buf.base);
    free(wr);
}

void on_stdout_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void on_file_write(uv_write_t *req, int status) {
    free_write_req(req);
}

void write_data(uv_stream_t *dest, size_t size, uv_buf_t buf, uv_wr
    write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
    req->buf = uv_buf_init((char*) malloc(size), size);
    memcpy(req->buf.base, buf.base, size);
    uv_write((uv_write_t*) req, (uv_stream_t*)dest, &req->buf, 1, c
}
```

`write_data()` 开辟了一块地址空间存储从缓冲区读取出来的数据，这块缓存不会被释放，直到与 `uv_write()` 绑定的回调函数执行.为了实现它，我们用结构体 `write_req_t` 包裹一个write request和一个buffer，然后在回调函数中展开它。因为我们复制了一份缓存，所以我们可以在两个 `write_data()` 中独立释放两个缓存。 我们之所以这样做是因为，两个调用 `write_data()` 是相互独立的。为了保证它们不会因为读取速度的原因，由于共享一片缓冲区而损失掉独立性，所以才开辟了新的两块区域。当然这只是一个简单的例子，你可以使用更聪明的内存管理方法来实现它，比如引用计数或者缓冲区池等。

WARNING

> 你的程序在被其他的程序调用的过程中，有意无意地会向pipe写入数据，这样的话它会很容易被信号SIGPIPE终止掉，你最好在初始化程序的时候加入这句：
> `signal(SIGPIPE, SIG_IGN)`。

File change events

所有的现代操作系统都会提供相应的API来监视文件和文件夹的变化(如**Linux**的**inotify**，**Darwin**的**FSEvents**，**BSD**的**kqueue**，**Windows**的**ReadDirectoryChangesW**， **Solaris**的**event ports**)。libuv同样包括了这样的文件

监视库。这是libuv中很不协调的部分，因为在跨平台的前提上，实现这个功能很难。为了更好地说明，我们现在来写一个监视文件变化的命令：

```
./onchange <command> <file1> [file2] ...
```

实现这个监视器，要从 `uv_fs_event_init()` 开始：

onchange/main.c - The setup

```
int main(int argc, char **argv) {
    if (argc <= 2) {
        fprintf(stderr, "Usage: %s <command> <file1> [file2 ...]\n'
        return 1;
    }

    loop = uv_default_loop();
    command = argv[1];

    while (argc-- > 2) {
        fprintf(stderr, "Adding watch on %s\n", argv[argc]);
        uv_fs_event_t *fs_event_req = malloc(sizeof(uv_fs_event_t)
        uv_fs_event_init(loop, fs_event_req);
        // The recursive flag watches subdirectories too.
        uv_fs_event_start(fs_event_req, run_command, argv[argc], UV
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

函数 `uv_fs_event_start()` 的第三个参数是要监视的文件或文件夹。最后一个参数， `flags` ，可以是：

```
  UV_FS_EVENT_WATCH_ENTRY = 1,
  UV_FS_EVENT_STAT = 2,
  UV_FS_EVENT_RECURSIVE = 4
```

`UV_FS_EVENT_WATCH_ENTRY` 和 `UV_FS_EVENT_STAT` 不做任何事情(至少目前是这样)， `UV_FS_EVENT_RECURSIVE` 可以在支持的系统平台上递归地监视子文件夹。
在回调函数 `run_command()` 中，接收的参数如下：

> 1. `uv_fs_event_t *handle` -句柄。里面的path保存了发生改变的文件的地址。
> 2. `const char *filename` -如果目录被监视，它代表发生改变的文件名。只在Linux和Windows上不为null，在其他平台上可能为null。
> 3. `int flags` - `UV_RENAME` 名字改变， `UV_CHANGE` 内容改变之一，或者他们两者的按位或的结果(`|`)。
> 4. `int status` －当前为0.

在我们的例子中，只是简单地打印参数和调用 `system()` 运行command.

onchange/main.c - file change notification callback

```
void run_command(uv_fs_event_t *handle, const char *filename, int e
    char path[1024];
    size_t size = 1023;
    // Does not handle error if path is longer than 1023.
    uv_fs_event_getpath(handle, path, &size);
    path[size] = '\0';

    fprintf(stderr, "Change detected in %s: ", path);
    if (events & UV_RENAME)
        fprintf(stderr, "renamed");
    if (events & UV_CHANGE)
        fprintf(stderr, "changed");

    fprintf(stderr, " %s\n", filename ? filename : "");
    system(command);
}
```

Networking

在libuv中使用网络编程接口不会像在BSD上使用socket接口那么的麻烦，因为libuv上所有的都是非阻塞的，但是原理都是一样的。可以这么说，libuv提供了覆盖了恼人的，啰嗦的和底层的任务的抽象函数，比如使用BSD的socket结构的来设置socket，还有DNS查找，libuv还调整了一些socket的参数。

在网络I/O中会使用到 `uv_tcp_t` 和 `uv_udp_t` 。

TCP

TCP是面向连接的，字节流协议，因此基于libuv的stream实现。

server

服务器端的建立流程如下：

> 1. `uv_tcp_init` 建立tcp句柄。
> 2. `uv_tcp_bind` 绑定。
> 3. `uv_listen` 建立监听，当有新的连接到来时，激活调用回调函数。
> 4. `uv_accept` 接收链接。
> 5. 使用stream处理来和客户端通信。

tcp-echo-server/main.c - The listen socket

```
int main() {
    loop = uv_default_loop();

    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    uv_ip4_addr("0.0.0.0", DEFAULT_PORT, &addr);

    uv_tcp_bind(&server, (const struct sockaddr*)&addr, 0);
    int r = uv_listen((uv_stream_t*) &server, DEFAULT_BACKLOG, on_n
    if (r) {
        fprintf(stderr, "Listen error %s\n", uv_strerror(r));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

你可以调用 `uv_ip4_addr()` 函数来将ip地址和端口号转换为sockaddr_in结构，这样就可以被BSD的socket使用了。要想完成逆转换的话可以调用 `uv_ip4_name()` 。

note

> 对应ipv6有类似的uv*ip6**

大多数的设置函数是同步的，因为它们不会消耗太多cpu资源。到了 `uv_listen` 这句，我们再次回到回调函数的风格上来。第二个参数是待处理的连接请求队列—最大长度的请求连接队列。

当客户端开始建立连接的时候，回调函数 `on_new_connection` 需要使用 `uv_accept` 去建立一个与客户端socket通信的句柄。同时，我们也要开始从流中读取数据。

tcp-echo-server/main.c - Accepting the client

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status < 0) {
        fprintf(stderr, "New connection error %s\n", uv_strerror(st
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_rea
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

上述的函数集和stream的例子类似，在code文件夹中可以找到更多的例子。记得在socket不需要后，调用uv_close。如果你不需要接受连接，你甚至可以在uv_listen的回调函数中调用uv_close。

client

当你在服务器端完成绑定／监听／接收的操作后，在客户端只要简单地调用 `uv_tcp_connect` ，它的回调函数和上面类似，具体例子如下：

```
uv_tcp_t* socket = (uv_tcp_t*)malloc(sizeof(uv_tcp_t));
uv_tcp_init(loop, socket);

uv_connect_t* connect = (uv_connect_t*)malloc(sizeof(uv_connect_t)

struct sockaddr_in dest;
uv_ip4_addr("127.0.0.1", 80, &dest);

uv_tcp_connect(connect, socket, dest, on_connect);
```

当建立连接后，回调函数 `on_connect` 会被调用。回调函数会接收到一个uv_connect_t结构的数据，它的 `handle` 指向通信的socket。

UDP

用户数据报协议(User Datagram Protocol)提供无连接的，不可靠的网络通信。因此，libuv不会提供一个stream实现的形式，而是提供了一个 `uv_udp_t` 句柄（接收端），和一个 `uv_udp_send_t` 句柄（发送端），还有相关的函数。也就是说，实际的读写api与正常的流读取类似。下面的例子展示了一个从DCHP服务器获取ip的例子。

note

> 你必须以管理员的权限运行udp-dhcp，因为它的端口号低于1024

udp-dhcp/main.c - Setup and send UDP packets

```
uv_loop_t *loop;
uv_udp_t send_socket;
uv_udp_t recv_socket;

int main() {
    loop = uv_default_loop();

    uv_udp_init(loop, &recv_socket);
    struct sockaddr_in recv_addr;
    uv_ip4_addr("0.0.0.0", 68, &recv_addr);
    uv_udp_bind(&recv_socket, (const struct sockaddr *)&recv_addr,
    uv_udp_recv_start(&recv_socket, alloc_buffer, on_read);

    uv_udp_init(loop, &send_socket);
    struct sockaddr_in broadcast_addr;
    uv_ip4_addr("0.0.0.0", 0, &broadcast_addr);
    uv_udp_bind(&send_socket, (const struct sockaddr *)&broadcast_a
    uv_udp_set_broadcast(&send_socket, 1);

    uv_udp_send_t send_req;
    uv_buf_t discover_msg = make_discover_msg();

    struct sockaddr_in send_addr;
    uv_ip4_addr("255.255.255.255", 67, &send_addr);
    uv_udp_send(&send_req, &send_socket, &discover_msg, 1, (const s

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

note

> ip地址为0.0.0.0，用来绑定所有的接口。255.255.255.255是一个广播地址，这也意味着数据报将往所有的子网接口中发送。端口号为0代表着由操作系统随机分配一个端口。

首先，我们设置了一个接收的socket，端口号为68，作为DHCP客户端，然后开始从中读取数据。它会接收所有来自DHCP服务器的返回数据。我们设置了 `UV_UDP_REUSEADDR` 标记，用来和其他共享端口的 DHCP客户端和平共处。接着，我们设置了一个类似的发送socket，然后使用 `uv_udp_send` 向DHCP服务器（在67端口）发送广播。

设置广播发送是非常必要的，否则你会接收到 `EACCES` 错误。和此前一样，如果在读写中出错，返回码<0。

因为UDP不会建立连接，因此回调函数会接收到关于发送者的额外的信息。

当没有可读数据后，nread等于0。如果 `addr` 是 `null` ，它代表了没有可读数据（回调函数不会做任何处理）。如果不为null，则说明了从addr中接收到一个空的数据报。如果flag为 `UV_UDP_PARTIAL` ，则代表了内存分配的空间不够存放接收到的数据了，在这种情形下，操作系统会丢弃存不下的数据。

udp-dhcp/main.c - Reading packets

```
void on_read(uv_udp_t *req, ssize_t nread, const uv_buf_t *buf, cor
    if (nread < 0) {
        fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        uv_close((uv_handle_t*) req, NULL);
        free(buf->base);
        return;
    }

    char sender[17] = { 0 };
    uv_ip4_name((const struct sockaddr_in*) addr, sender, 16);
    fprintf(stderr, "Recv from %s\n", sender);

    // ... DHCP specific code
    unsigned int *as_integer = (unsigned int*)buf->base;
    unsigned int ipbin = ntohl(as_integer[4]);
    unsigned char ip[4] = {0};
    int i;
    for (i = 0; i < 4; i++)
        ip[i] = (ipbin >> i*8) & 0xff;
    fprintf(stderr, "Offered IP %d.%d.%d.%d\n", ip[3], ip[2], ip[1]

    free(buf->base);
    uv_udp_recv_stop(req);
}
```

UDP Options

生存时间（Time-to-live）

> 可以通过 `uv_udp_set_ttl` 更改生存时间。

只允许IPV6协议栈

> 在调用 `uv_udp_bind` 时，设置 `UV_UDP_IPV6ONLY` 标示，可以强制只使用ipv6。

组播

> socket也支持组播，可以这么使用：

```
UV_EXTERN int uv_udp_set_membership(uv_udp_t* handle,
                                    const char* multicast_addr,
                                    const char* interface_addr,
                                    uv_membership membership);
```

其中 `membership` 可以为 `UV_JOIN_GROUP` 和 `UV_LEAVE_GROUP` 。
这里有一篇很好的关于组播的文章。
可以使用 `uv_udp_set_multicast_loop` 修改本地的组播。
同样可以使用 `uv_udp_set_multicast_ttl` 修改组播数据报的生存时间。（设定生存时间可以防止数据报由于环路的原因，会出现无限循环的问题）。

Querying DNS

libuv提供了一个异步的DNS解决方案。它提供了自己的 `getaddrinfo` 。在回调函数中你可以像使用正常的socket操作一样。让我们来看一下例子：

dns/main.c

```
int main() {
    loop = uv_default_loop();

    struct addrinfo hints;
    hints.ai_family = PF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    uv_getaddrinfo_t resolver;
    fprintf(stderr, "irc.freenode.net is... ");
    int r = uv_getaddrinfo(loop, &resolver, on_resolved, "irc.freen

    if (r) {
        fprintf(stderr, "getaddrinfo call error %s\n", uv_err_name(
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

如果 `uv_getaddrinfo` 返回非零值，说明设置错误了，因此也不会激发回调函数。在函数返回后，所有的参数将会被回收和释放。主机地址，请求服务器地址，还有hints的结构都可以在这里找到详细的说明。如果想使用同步请求，可以将回调函数设置为NULL。

在回调函数on_resolved中，你可以从 `struct addrinfo(s)` 链表中获取返回的IP，最后需要调用 `uv_freeaddrinfo` 回收掉链表。下面的例子演示了回调函数的内容。

dns/main.c

```
void on_resolved(uv_getaddrinfo_t *resolver, int status, struct add
    if (status < 0) {
        fprintf(stderr, "getaddrinfo callback error %s\n", uv_err_n
        return;
    }

    char addr[17] = {'\0'};
    uv_ip4_name((struct sockaddr_in*) res->ai_addr, addr, 16);
    fprintf(stderr, "%s\n", addr);

    uv_connect_t *connect_req = (uv_connect_t*) malloc(sizeof(uv_co
    uv_tcp_t *socket = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, socket);

    uv_tcp_connect(connect_req, socket, (const struct sockaddr*) re

    uv_freeaddrinfo(res);
}
```

libuv同样提供了DNS逆解析的函数uv_getnameinfo。

Network interfaces

可以调用 `uv_interface_addresses` 获得系统的网络接口信息。下面这个简单的例子打印出所有可以获取的信息。这在服务器开始准备绑定IP地址的时候很有用。

interfaces/main.c

```
#include <stdio.h>
#include <uv.h>

int main() {
    char buf[512];
    uv_interface_address_t *info;
    int count, i;

    uv_interface_addresses(&info, &count);
    i = count;

    printf("Number of interfaces: %d\n", count);
    while (i--) {
        uv_interface_address_t interface = info[i];

        printf("Name: %s\n", interface.name);
        printf("Internal? %s\n", interface.is_internal ? "Yes" : "N

        if (interface.address.address4.sin_family == AF_INET) {
            uv_ip4_name(&interface.address.address4, buf, sizeof(bu
            printf("IPv4 address: %s\n", buf);
        }
        else if (interface.address.address4.sin_family == AF_INET6)
            uv_ip6_name(&interface.address.address6, buf, sizeof(bu
            printf("IPv6 address: %s\n", buf);
        }

        printf("\n");
    }

    uv_free_interface_addresses(info, count);
    return 0;
}
```

`is_internal` 可以用来表示是否是内部的IP。由于一个物理接口会有多个IP地址，所以每一次while循环的时候都会打印一次。

Threads

等一下！为什么我们要聊线程？事件循环（event loop）不应该是用来做web编程的方法吗？(如果你对event loop, 不是很了解，可以看这里)。哦，不不。线程依旧是处理器完成任务的重要手段。线程因此有可能会派上用场，虽然会使得你不得不艰难地应对各种原始的同步问题。

线程会在内部使用，用来在执行系统调用时伪造异步的假象。libuv通过线程还可以使得程序异步地执行一个阻塞的任务。方法就是大量地生成新线程，然后收集线程执行返回的结果。

当下有两个占主导地位的线程库：windows下的线程实现和POSIX的pthread。libuv的线程API与pthread的API在使用方法和语义上很接近。

值得注意的是，libuv的线程模块是自成一体的。比如，其他的功能模块都需要依赖于event loop和回调的原则，但是线程并不是这样。它们是不受约束的，会在需要的时候阻塞，通过返回值产生信号错误，还有像接下来的这个例子所演示的这样，不需要在event loop中执行。

因为线程API在不同的系统平台上，句法和语义表现得都不太相似，在支持程度上也各不相同。考虑到libuv的跨平台特性，libuv支持的线程API个数很有限。

最后要强调一句：只有一个主线程，主线程上只有一个event loop。不会有其他与主线程交互的线程了。（除非使用`uv_async_send`）。

Core thread operations

下面这个例子不会很复杂，你可以使用`uv_thread_create()`开始一个线程，再使用`uv_thread_join()`等待其结束。

thread-create/main.c

```
int main() {
    int tracklen = 10;
    uv_thread_t hare_id;
    uv_thread_t tortoise_id;
    uv_thread_create(&hare_id, hare, &tracklen);
    uv_thread_create(&tortoise_id, tortoise, &tracklen);

    uv_thread_join(&hare_id);
    uv_thread_join(&tortoise_id);
    return 0;
}
```

TIP

> 在Unix上 `uv_thread_t` 只是 `pthread_t` 的别名, 但是这只是一个具体实现，不要过度地依赖它，认为这永远是成立的。

`uv_thread_t` 的第二个参数指向了要执行的函数的地址。最后一个参数用来传递自定义的参数。最终，函数hare将在新的线程中执行，由操作系统调度。

thread-create/main.c

```
void hare(void *arg) {
    int tracklen = *((int *) arg);
    while (tracklen) {
        tracklen--;
        sleep(1);
        fprintf(stderr, "Hare ran another step\n");
    }
    fprintf(stderr, "Hare done running!\n");
}
```

`uv_thread_join` 不像 `pthread_join` 那样，允许线线程通过第二个参数向父线程返回值。想要传递值，必须使用线程间通信Inter-thread communication。

Synchronization Primitives

因为本教程重点不在线程，所以我只罗列了libuv API中一些神奇的地方。剩下的你可以自行阅读pthreads的手册。

Mutexes

libuv上的互斥量函数与pthread上存在一一映射。如果对pthread上的mutex不是很了解可以看这里。

libuv mutex functions

```
UV_EXTERN int uv_mutex_init(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_destroy(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_lock(uv_mutex_t* handle);
UV_EXTERN int uv_mutex_trylock(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_unlock(uv_mutex_t* handle);
```

`uv_mutex_init` 与 `uv_mutex_trylock` 在成功执行后，返回0，或者在错误时，返回错误码。

如果libuv在编译的时候开启了调试模式， `uv_mutex_destroy()` , `uv_mutex_lock()` 和 `uv_mutex_unlock()` 会在出错的地方调用 `abort()` 中断。类似的， `uv_mutex_trylock()` 也同样会在错误发生时中断，而不是返回 `EAGAIN` 和 `EBUSY` 。

递归地调用互斥量函数在某些系统平台上是支持的，但是你不能太过度依赖。因为例如在BSD上递归地调用互斥量函数会返回错误，比如你准备使用互斥量函数给一个已经上锁的临界区再次上锁的时候，就会出错。比如，像下面这个例子：

```
uv_mutex_lock(a_mutex);
uv_thread_create(thread_id, entry, (void *)a_mutex);
uv_mutex_lock(a_mutex);
// more things here
```

可以用来等待其他线程初始化一些变量然后释放 `a_mutex` 锁，但是第二次调用 `uv_mutex_lock()` , 在调试模式下会导致程序崩溃，或者是返回错误。

NOTE

> 在linux中是支持递归上锁的，但是在libuv的API中并未实现。

Lock

读写锁是更细粒度的实现机制。两个读者线程可以同时从共享区中读取数据。当读者以读模式占有读写锁时，写者不能再占有它。当写者以写模式占有这个锁时，其他的写者或者读者都不能占有它。读写锁在数据库操作中非常常见，下面是一个玩具式的例子：

ocks/main.c - simple rwlocks

```
#include <stdio.h>
#include <uv.h>

uv_barrier_t blocker;
uv_rwlock_t numlock;
int shared_num;

void reader(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_rdlock(&numlock);
        printf("Reader %d: acquired lock\n", num);
        printf("Reader %d: shared num = %d\n", num, shared_num);
        uv_rwlock_rdunlock(&numlock);
        printf("Reader %d: released lock\n", num);
```

```
    }
    uv_barrier_wait(&blocker);
}

void writer(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_wrlock(&numlock);
        printf("Writer %d: acquired lock\n", num);
        shared_num++;
        printf("Writer %d: incremented shared num = %d\n", num, sha
        uv_rwlock_wrunlock(&numlock);
        printf("Writer %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

int main()
{
    uv_barrier_init(&blocker, 4);

    shared_num = 0;
    uv_rwlock_init(&numlock);

    uv_thread_t threads[3];

    int thread_nums[] = {1, 2, 1};
    uv_thread_create(&threads[0], reader, &thread_nums[0]);
    uv_thread_create(&threads[1], reader, &thread_nums[1]);

    uv_thread_create(&threads[2], writer, &thread_nums[2]);

    uv_barrier_wait(&blocker);
    uv_barrier_destroy(&blocker);

    uv_rwlock_destroy(&numlock);
    return 0;
}
```

试着来执行一下上面的程序，看读者有多少次会同步执行。在有多个写者的时候，调度器会给予他们高优先级。因此，如果你加入两个读者，你会看到所有的读者趋向于在读者得到加锁机会前结束。

在上面的例子中，我们也使用了屏障。因此主线程来等待所有的线程都已经结束，最后再将屏障和锁一块回收。

Others

libuv同样支持信号量，条件变量和屏障，而且API的使用方法和pthread中的用法很类似。（如果你对上面的三个名词还不是很熟，可以看这里，这里，这里）。

还有，libuv提供了一个简单易用的函数 `uv_once()` 。多个线程调用这个函数，参数可以使用一个uv_once_t和一个指向特定函数的指针，最终只有一个线程能够执行这个特定函数，并且这个特定函数只会被调用一次：

```
/* Initialize guard */
static uv_once_t once_only = UV_ONCE_INIT;

int i = 0;

void increment() {
    i++;
}

void thread1() {
    /* ... work */
    uv_once(once_only, increment);
}

void thread2() {
    /* ... work */
    uv_once(once_only, increment);
}

int main() {
    /* ... spawn threads */
}
```

当所有的线程执行完毕时， `i == 1` 。

在libuv的v0.11.11版本里，推出了uv_key_t结构和操作线程局部存储TLS的API，使用方法同样和pthread类似。

libuv work queue

`uv_queue_work()` 是一个便利的函数，它使得一个应用程序能够在不同的线程运行任务，当任务完成后，回调函数将会被触发。它看起来好像很简单，但是它真正吸引人的地方在于它能够使得任何第三方的库都能以event-loop的方式执行。当使用event-loop的时候，最重要的是不能让loop线程阻塞，或者是执行高cpu占用的程序，因为这样会使得loop慢下来，loop event的高效特性也不能得到很好地发挥。

然而，很多带有阻塞的特性的程序(比如最常见的I/O）使用开辟新线程来响应新请求(最经典的‘一个客户，一个线程‘模型)。使用event-loop可以提供另一种实现的方式。libuv提供了一个很好的抽象，使得你能够很好地使用它。

下面有一个很好的例子，灵感来自<<nodejs is cancer>>。我们将要执行fibonacci数列，并且睡眠一段时间，但是将阻塞和cpu占用时间长的任务分配到不同的线程，使得其不会阻塞event loop上的其他任务。

queue-work/main.c - lazy fibonacci

```
void fib(uv_work_t *req) {
    int n = *(int *) req->data;
    if (random() % 2)
        sleep(1);
    else
        sleep(3);
    long fib = fib_(n);
    fprintf(stderr, "%dth fibonacci is %lu\n", n, fib);
}

void after_fib(uv_work_t *req, int status) {
    fprintf(stderr, "Done calculating %dth fibonacci\n", *(int *) 
}
```

任务函数很简单，也还没有运行在线程之上。`uv_work_t`是关键线索，你可以通过`void *data`传递任何数据，使用它来完成线程之间的沟通任务。但是你要确信，当你在多个线程都在运行的时候改变某个东西的时候，能够使用适当的锁。

触发器是`uv_queue_work`：

queue-work/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

线程函数fbi()将会在不同的线程中运行，传入`uv_work_t`结构体参数，一旦fib()函数返回，after_fib()会被event loop中的线程调用，然后被传入同样的结构体。

为了封装阻塞的库，常见的模式是用baton来交换数据。

从libuv 0.9.4版后，添加了函数 `uv_cancel()` 。它可以用来取消工作队列中的任务。只有还未开始的任务可以被取消，如果任务已经开始执行或者已经执行完毕， `uv_cancel()` 调用会失败。

当用户想要终止程序的时候， `uv_cancel()` 可以用来清理任务队列中的等待执行的任务。例如，一个音乐播放器可以以歌手的名字对歌曲进行排序，如果这个时候用户想要退出这个程序， `uv_cancel()` 就可以做到快速退出，而不用等待执行完任务队列后，再退出。

让我们对上述程序做一些修改，用来演示 `uv_cancel()` 的用法。首先让我们注册一个处理中断的函数。

queue-cancel/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        fib_reqs[i].data = (void *) &data[i];
        uv_queue_work(loop, &fib_reqs[i], fib, after_fib);
    }

    uv_signal_t sig;
    uv_signal_init(loop, &sig);
    uv_signal_start(&sig, signal_handler, SIGINT);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

当用户通过 `Ctrl+C` 触发信号时， `uv_cancel()` 回收任务队列中所有的任务，如果任务已经开始执行或者执行完毕， `uv_cancel()` 返回0。

queue-cancel/main.c

```
void signal_handler(uv_signal_t *req, int signum)
{
    printf("Signal received!\n");
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        uv_cancel((uv_req_t*) &fib_reqs[i]);
    }
    uv_signal_stop(req);
}
```

对于已经成功取消的任务，他的回调函数的参数 `status` 会被设置为 `UV_ECANCELED` 。

queue-cancel/main.c

```
void after_fib(uv_work_t *req, int status) {
    if (status == UV_ECANCELED)
        fprintf(stderr, "Calculation of %d cancelled.\n", *(int *)
}
```

`uv_cancel()` 函数同样可以用在 `uv_fs_t` 和 `uv_getaddrinfo_t` 请求上。对于一系列的文件系统操作函数来说， `uv_fs_t.errorno` 会同样被设置为 `UV_ECANCELED` 。

Tip

> 一个良好设计的程序，应该能够终止一个已经开始运行的长耗时任务。
> Such a worker could periodically check for a variable that only the main process sets to signal termination.

Inter-thread communication

很多时候，你希望正在运行的线程之间能够相互发送消息。例如你在运行一个持续时间长的任务（可能使用uv_queue_work），但是你需要在主线程中监视它的进度情况。下面有一个简单的例子，演示了一个下载管理程序向用户展示各个下载线程的进度。

progress/main.c

```
uv_loop_t *loop;
uv_async_t async;

int main() {
    loop = uv_default_loop();

    uv_work_t req;
    int size = 10240;
    req.data = (void*) &size;

    uv_async_init(loop, &async, print_progress);
    uv_queue_work(loop, &req, fake_download, after);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

因为异步的线程通信是基于event-loop的，所以尽管所有的线程都可以是发送方，但是只有在event-loop上的线程可以是接收方（或者说event-loop是接收方）。在上述的代码中，当异步监视者接收到信号的时候，libuv会激发回调函数（print_progress）。

WARNING

> 应该注意: 因为消息的发送是异步的,当 `uv_async_send` 在另外一个线程中被调用后，回调函数可能会立即被调用, 也可能在稍后的某个时刻被调用。libuv也有可能多次调用 `uv_async_send` ，但只调用了一次回调函数。唯一可以保证的是: 线程在调用 `uv_async_send` 之后回调函数可至少被调用一次。如果你没有未调用的 `uv_async_send` , 那么回调函数也不会被调用。如果你调用了两次(以上)的 `uv_async_send` , 而 libuv 暂时还没有机会运行回调函数, 则libuv可能会在多次调用 `uv_async_send` 后只调用一次回调函数，你的回调函数绝对不会在一次事件中被调用两次(或多次)。

progress/main.c

```
void fake_download(uv_work_t *req) {
    int size = *((int*) req->data);
    int downloaded = 0;
    double percentage;
    while (downloaded < size) {
        percentage = downloaded*100.0/size;
        async.data = (void*) &percentage;
        uv_async_send(&async);

        sleep(1);
        downloaded += (200+random())%1000; // can only download max
                                           // but at least a 200;
    }
}
```

在上述的下载函数中，我们修改了进度显示器，使用 `uv_async_send` 发送进度信息。要记住： `uv_async_send` 同样是非阻塞的，调用后会立即返回。

progress/main.c

```
void print_progress(uv_async_t *handle) {
    double percentage = *((double*) handle->data);
    fprintf(stderr, "Downloaded %.2f%%\n", percentage);
}
```

函数 `print_progress` 是标准的libuv模式，从监视器中抽取数据。

最后最重要的是把监视器回收。

progress/main.c

```
void after(uv_work_t *req, int status) {
    fprintf(stderr, "Download complete\n");
    uv_close((uv_handle_t*) &async, NULL);
}
```

在例子的最后，我们要说下 `data` 域的滥用，bnoordhuis指出使用 `data` 域可能会存在线程安全问题， `uv_async_send()` 事实上只是唤醒了event-loop。可以使用互斥量或者读写锁来保证执行顺序的正确性。

Note

> 互斥量和读写锁不能在信号处理函数中正确工作，但是 `uv_async_send` 可以。

一种需要使用 `uv_async_send` 的场景是，当调用需要线程交互的库时。例如，举一个在node.js中V8引擎的例子，上下文和对象都是与v8引擎的线程绑定的，从另一个线程中直接向v8请求数据会导致返回不确定的结果。但是，考虑到现在很多nodejs的模块都是和第三方库绑定的，可以像下面一样，解决这个问题：

> 1.在node中，第三方库会建立javascript的回调函数，以便回调函数被调用时，能够返回更多的信息。

```
var lib = require('lib');
lib.on_progress(function() {
    console.log("Progress");
});

lib.do();

// do other stuff
```

> 2. `lib.do` 应该是非阻塞的，但是第三方库却是阻塞的，所以需要调用 `uv_queue_work` 函数。
> 3.在另外一个线程中完成任务想要调用progress的回调函数，但是不能直接与v8通信，所以需要 `uv_async_send` 函数。
> 4.在主线程（v8线程）中调用的异步回调函数，会在v8的配合下执行javscript的回调函数。（也就是说，主线程会调用回调函数，并且提供v8解析javascript的功能，以便其完成任务）。

Processes

libuv提供了相当多的子进程管理函数，并且是跨平台的，还允许使用stream，或者说pipe完成进程间通信。

在UNIX中有一个共识，就是进程只做一件事，并把它做好。因此，进程通常通过创建子进程来完成不同的任务（例如，在shell中使用pipe）。 一个多进程的，通过消息通信的模型，总比多线程的，共享内存的模型要容易理解得多。

当前一个比较常见的反对事件驱动编程的原因在于，其不能很好地利用现代多核计算机的优势。一个多线程的程序，内核可以将线程调度到不同的cpu核心中执行，以提高性能。但是一个event-loop的程序只有一个线程。实际上，工作区可以被分配到多进程上，每一个进程执行一个event-loop，然后每一个进程被分配到不同的cpu核心中执行。

Spawning child processes

一个最简单的用途是，你想要开始一个进程，然后知道它什么时候终止。需要使用 `uv_spawn` 完成任务：

spawn/main.c

```
uv_loop_t *loop;
uv_process_t child_req;
uv_process_options_t options;
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "mkdir";
    args[1] = "test-dir";
    args[2] = NULL;

    options.exit_cb = on_exit;
    options.file = "mkdir";
    options.args = args;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    } else {
        fprintf(stderr, "Launched process with ID %d\n", child_req
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

Note

> 由于上述的options是全局变量，因此被初始化为0。如果你在局部变量中定义options，请记得将所有没用的域设为0

```
uv_process_options_t options = {0};
```

`uv_process_t` 只是作为句柄，所有的选择项都通过 `uv_process_options_t` 设置，为了简单地开始一个进程，你只需要设置file和args，file是要执行的程序，args是所需的参数（和c语言中main函数的传入参数类似）。因为 `uv_spawn` 在内部使用了execvp，所以不需要提供绝对地址。遵从惯例，实际传入参数的数目要比需要的参数多一个，因为最后一个参数会被设为**NULL**。

在函数 `uv_spawn` 被调用之后， `uv_process_t.pid` 会包含子进程的id。

回调函数 `on_exit()` 会在被调用的时候，传入exit状态和导致exit的信号。

spawn/main.c

```
void on_exit(uv_process_t *req, int64_t exit_status, int term_signa
    fprintf(stderr, "Process exited with status %" PRId64 ", signal
    uv_close((uv_handle_t*) req, NULL);
```

在进程关闭后，需要回收handler。

Changing process parameters

在子进程开始执行前，你可以通过使用 `uv_process_options_t` 设置运行环境。

Change execution directory

设置 `uv_process_options_t.cwd` ，更改相应的目录。

Set environment variables

`uv_process_options_t.env` 的格式是以null为结尾的字符串数组，其中每一个字符串的形式都是 `VAR=VALUE` 。这些值用来设置进程的环境变量。如果子进程想要继承父进程的环境变量，就将 `uv_process_options_t.env` 设为null。

Option flags

通过使用下面标识的按位或的值设置 `uv_process_options_t.flags` 的值，可以定义子进程的行为：

> - `UV_PROCESS_SETUID` -将子进程的执行用户id（UID）设置为 `uv_process_options_t.uid` 中的值。
> - `UV_PROCESS_SETGID` -将子进程的执行组id(GID)设置为 `uv_process_options_t.gid` 中的值。
>   只有在unix系的操作系统中支持设置用户id和组id，在windows下设置会失败， `uv_spawn` 会返回 `UV_ENOTSUP` 。
> - `UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS` -在windows上， `uv_process_options_t.args` 参数不要用引号包裹。此标记对unix无效。
> - `UV_PROCESS_DETACHED` -在新会话(session)中启动子进程，这样子进程就可以在父进程退出后继续进行。请看下面的例子：

Detaching processes

使用标识 `UV_PROCESS_DETACHED` 可以启动守护进程(daemon)，或者是使得子进程从父进程中独立出来，这样父进程的退出就不会影响到它。

detach/main.c

```
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "sleep";
    args[1] = "100";
    args[2] = NULL;

    options.exit_cb = NULL;
    options.file = "sleep";
    options.args = args;
    options.flags = UV_PROCESS_DETACHED;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    }
    fprintf(stderr, "Launched sleep with PID %d\n", child_req.pid);
    uv_unref((uv_handle_t*) &child_req);

    return uv_run(loop, UV_RUN_DEFAULT);
```

记住一点，就是handle会始终监视着子进程，所以你的程序不会退出。`uv_unref()` 会解除handle。

Sending signals to processes

libuv打包了unix标准的 `kill(2)` 系统调用，并且在windows上实现了一个类似用法的调用，但要注意：所有的 `SIGTERM` ， `SIGINT` 和 `SIGKILL` 都会导致进程的中断。 `uv_kill` 函数如下所示：

```
uv_err_t uv_kill(int pid, int signum);
```

对于用libuv启动的进程，应该使用 `uv_process_kill` 终止，它会以 `uv_process_t` 作为第一个参数，而不是pid。当使用 `uv_process_kill` 后，记得使用 `uv_close` 关闭 `uv_process_t` 。

Signals

libuv对unix信号和一些windows下类似的机制，做了很好的打包。

使用 `uv_signal_init` 初始化handle（ `uv_signal_t` ），然后将它与loop关联。为了使用handle监听特定的信号，使用 `uv_signal_start()` 函数。每一个handle只能与一个信号关联，后续的 `uv_signal_start` 会覆盖前面的关联。使用 `uv_signal_stop` 终止监听。下面的这个小例子展示了各种用法：

signal/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t* create_loop()
{
    uv_loop_t *loop = malloc(sizeof(uv_loop_t));
    if (loop) {
      uv_loop_init(loop);
    }
    return loop;
}

void signal_handler(uv_signal_t *handle, int signum)
{
    printf("Signal received: %d\n", signum);
    uv_signal_stop(handle);
}

// two signal handlers in one loop
void thread1_worker(void *userp)
{
    uv_loop_t *loop1 = create_loop();

    uv_signal_t sig1a, sig1b;
    uv_signal_init(loop1, &sig1a);
    uv_signal_start(&sig1a, signal_handler, SIGUSR1);

    uv_signal_init(loop1, &sig1b);
    uv_signal_start(&sig1b, signal_handler, SIGUSR1);

    uv_run(loop1, UV_RUN_DEFAULT);
}

// two signal handlers, each in its own loop
void thread2_worker(void *userp)
{
    uv_loop_t *loop2 = create_loop();
    uv_loop_t *loop3 = create_loop();

    uv_signal_t sig2;
    uv_signal_init(loop2, &sig2);
    uv_signal_start(&sig2, signal_handler, SIGUSR1);
```

```
    uv_signal_t sig3;
    uv_signal_init(loop3, &sig3);
    uv_signal_start(&sig3, signal_handler, SIGUSR1);

    while (uv_run(loop2, UV_RUN_NOWAIT) || uv_run(loop3, UV_RUN_NOW
    }
}

int main()
{
    printf("PID %d\n", getpid());

    uv_thread_t thread1, thread2;

    uv_thread_create(&thread1, thread1_worker, 0);
    uv_thread_create(&thread2, thread2_worker, 0);

    uv_thread_join(&thread1);
    uv_thread_join(&thread2);
    return 0;
}
```

Note

> `uv_run(loop, UV_RUN_NOWAIT)` 和 `uv_run(loop, UV_RUN_ONCE)` 非常像，因为它们都只处理一个事件。但是不同在于，UV_RUN_ONCE会在没有任务的时候阻塞，但是UV_RUN_NOWAIT会立刻返回。我们使用 `NOWAIT` ，这样才使得一个loop不会因为另外一个loop没有要处理的事件而挨饿。

当向进程发送 `SIGUSR1` ，你会发现signal_handler函数被激发了4次，每次都对应一个 `uv_signal_t` 。然后signal_handler调用uv_signal_stop终止了每一个 `uv_signal_t` ，最终程序退出。对每个handler函数来说，任务的分配很重要。一个使用了多个event-loop的服务器程序，只要简单地给每一个进程添加信号SIGINT监视器，就可以保证程序在中断退出前，数据能够安全地保存。

Child Process I/O

一个正常的新产生的进程都有自己的一套文件描述符映射表，例如0，1，2分别对应 `stdin` ， `stdout` 和 `stderr` 。有时候父进程想要将自己的文件描述符映射表分享给子进程。例如，你的程序启动了一个子命令，并且把所有的错误信息输出到log文件中，但是不能使用 `stdout` 。因此，你想要使得你的子进程和父进程一样，拥有 `stderr` 。在这种情形下，libuv提供了继承文件描述符的功能。在下面的例子中，我们会调用这么一个测试程序：

proc-streams/test.c

```
#include <stdio.h>

int main()
{
    fprintf(stderr, "This is stderr\n");
    printf("This is stdout\n");
    return 0;
}
```

实际的执行程序 `proc-streams` 在运行的时候，只向子进程分享 `stderr` 。使用 `uv_process_options_t` 的 `stdio` 域设置子进程的文件描述符。首先设置 `stdio_count` ，定义文件描述符的个数。 `uv_process_options_t.stdio` 是一个 `uv_stdio_container_t` 数组。定义如下：

```
typedef struct uv_stdio_container_s {
  uv_stdio_flags flags;

  union {
    uv_stream_t* stream;
    int fd;
  } data;
} uv_stdio_container_t;
```

上边的flag值可取多种。比如，如果你不打算使用，可以设置为 `UV_IGNORE` 。如果与stdio中对应的前三个文件描述符被标记为 `UV_IGNORE` ，那么它们会被重定向到 `/dev/null` 。

因为我们想要传递一个已经存在的文件描述符，所以使用 `UV_INHERIT_FD` 。因此，fd被设为stderr。

proc-streams/main.c

```
int main() {
    loop = uv_default_loop();

    /* ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_IGNORE;
    child_stdio[2].flags = UV_INHERIT_FD;
    child_stdio[2].data.fd = 2;
    options.stdio = child_stdio;

    options.exit_cb = on_exit;
    options.file = args[0];
    options.args = args;

    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
        return 1;
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

这时你启动proc-streams，也就是在main中产生一个执行test的子进程，你只会看到“This is stderr”。你可以试着设置stdout也继承父进程。

同样可以把上述方法用于流的重定向。比如，把flag设为 `UV_INHERIT_STREAM`，然后再设置父进程中的 `data.stream`，这时子进程只会把这个stream当成是标准的I/O。这可以用来实现，例如CGI。

一个简单的CGI脚本的例子如下：

cgi/tick.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("tick\n");
        fflush(stdout);
        sleep(1);
    }
    printf("BOOM!\n");
    return 0;
}
```

CGI服务器用到了这章和网络那章的知识，所以每一个client在中断连接后，都会被发送10个tick。

cgi/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        invoke_cgi_script(client);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
```

上述代码中，我们接受了连接，并把socket（流）传递给 `invoke_cgi_script` 。

cgi/main.c

```
args[1] = NULL;

    /* ... finding the executable path and setting up arguments ...

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_INHERIT_STREAM;
    child_stdio[1].data.stream = (uv_stream_t*) client;
    child_stdio[2].flags = UV_IGNORE;
    options.stdio = child_stdio;

    options.exit_cb = cleanup_handles;
    options.file = args[0];
    options.args = args;

    // Set this so we can close the socket after the child process
    child_req.data = (void*) client;
    int r;
    if ((r = uv_spawn(loop, &child_req, &options))) {
        fprintf(stderr, "%s\n", uv_strerror(r));
```

cgi的 `stdout` 被绑定到socket上，所以无论tick脚本程序打印什么，都会发送到client端。通过使用进程，我们能够很好地处理读写并发操作，而且用起来也很方便。但是要记得这么做，是很浪费资源的。

Pipes

libuv的 `uv_pipe_t` 结构可能会让一些unix程序员产生困惑，因为它像魔术般变幻出 `|` 和 `pipe(7)` 。但这里的 `uv_pipe_t` 并不是IPC机制里的匿名管道（在IPC里，pipe是匿名管道，只允许父子进程之间通信。FIFO则允许没有亲戚关系的进程间通信，显然llibuv里的 `uv_pipe_t` 不是第一种）。 `uv_pipe_t` 背后有unix本地socket或者windows实名管道的支持，可以实现多进程间的通信。下面会具体讨论。

Parent-child IPC

父进程与子进程可以通过单工或者双工管道通信，获得管道可以通过设置 `uv_stdio_container_t.flags` 为 `UV_CREATE_PIPE` ， `UV_READABLE_PIPE` 或者 `UV_WRITABLE_PIPE` 的按位或的值。上述的读／写标记是对于子进程而言的。

Arbitrary process IPC

既然本地socket具有确定的名称，而且是以文件系统上的位置来标示的（例如，unix中socket是文件的一种存在形式），那么它就可以用来在不相关的进程间完成通信任务。被开源桌面环境使用的 D-BUS 系统也是使用了本地socket来作为事件通知的，例如，当消息来到，或者检测到硬件的时候，各种应用程序会被通知到。mysql服务器也运行着一个本地socket，等待客户端的访问。

当使用本地socket的时候，客户端／服务器模型通常和之前类似。在完成初始化后，发送和接受消息的方法和之前的tcp类似，接下来我们同样适用echo服务器的例子来说明。

pipe-echo-server/main.c

```
int main() {
    loop = uv_default_loop();

    uv_pipe_t server;
    uv_pipe_init(loop, &server, 0);

    signal(SIGINT, remove_sock);

    int r;
    if ((r = uv_pipe_bind(&server, "echo.sock"))) {
        fprintf(stderr, "Bind error %s\n", uv_err_name(r));
        return 1;
    }
    if ((r = uv_listen((uv_stream_t*) &server, 128, on_new_connect
        fprintf(stderr, "Listen error %s\n", uv_err_name(r));
        return 2;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

我们把socket命名为echo.sock，意味着它将会在本地文件夹中被创造。对于stream API来说，本地socekt表现得和tcp的socket差不多。你可以使用socat测试一下服务器：

```
$ socat - /path/to/socket
```

客户端如果想要和服务器端连接的话，应该使用：

```
void uv_pipe_connect(uv_connect_t *req, uv_pipe_t *handle, const ch
```

上述函数，name应该为echo.sock。

Sending file descriptors over pipes

最酷的事情是本地socket可以传递文件描述符，也就是说进程间可以交换文件描述符。这样就允许进程将它们的I/O传递给其他进程。它的应用场景包括，负载均衡服务器，分派工作进程等，各种可以使得cpu使用最优化的应用。libuv当前只支持通过管道传输**TCP sockets**或者其他的**pipes**。

为了展示这个功能，我们将来实现一个由循环中的工人进程处理client端请求，的这么一个echo服务器程序。这个程序有一些复杂，在教程中只截取了部分的片段，为了更好地理解，我推荐你去读下完整的代码。

工人进程很简单，文件描述符将从主进程传递给它。

multi-echo-server/worker.c

```
uv_loop_t *loop;
uv_pipe_t queue;
int main() {
    loop = uv_default_loop();

    uv_pipe_init(loop, &queue, 1 /* ipc */);
    uv_pipe_open(&queue, 0);
    uv_read_start((uv_stream_t*)&queue, alloc_buffer, on_new_connec
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

`queue` 是另一端连接上主进程的管道，因此，文件描述符可以传送过来。在 `uv_pipe_init` 中将 `ipc` 参数设置为1很关键，因为它标明了这个管道将被用来做进程间通信。因为主进程需要把文件handle赋给了工人进程作为标准输入，因此我们使用 `uv_pipe_open` 把stdin作为pipe（别忘了，0代表stdin）。

multi-echo-server/worker.c

```
void on_new_connection(uv_stream_t *q, ssize_t nread, const uv_buf_
    if (nread < 0) {
        if (nread != UV_EOF)
            fprintf(stderr, "Read error %s\n", uv_err_name(nread));
        uv_close((uv_handle_t*) q, NULL);
        return;
    }

    uv_pipe_t *pipe = (uv_pipe_t*) q;
    if (!uv_pipe_pending_count(pipe)) {
        fprintf(stderr, "No pending count\n");
        return;
    }

    uv_handle_type pending = uv_pipe_pending_type(pipe);
    assert(pending == UV_TCP);

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(q, (uv_stream_t*) client) == 0) {
        uv_os_fd_t fd;
        uv_fileno((const uv_handle_t*) client, &fd);
        fprintf(stderr, "Worker %d: Accepted fd %d\n", getpid(), fd
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_rea
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

首先，我们调用 `uv_pipe_pending_count` 来确定从handle中可以读取出数据。如果你的程序能够处理不同类型的handle，这时 `uv_pipe_pending_type` 就可以用来决定当前的类型。虽然在这里使用 `accept` 看起来很怪，但实际上是讲得通的。 `accept` 最常见的用途是从其他的文件描述符（监听的socket）获取文件描述符（client端）。这从原理上说，和我们现在要做的是一样的：从queue中获取文件描述符（client）。接下来，worker可以执行标准的echo服务器的工作了。

我们再来看看主进程，观察如何启动worker来达到负载均衡。

multi-echo-server/main.c

```
struct child_worker {
    uv_process_t req;
    uv_process_options_t options;
    uv_pipe_t pipe;
} *workers;
```

`child_worker` 结构包裹着进程，和连接主进程和各个独立进程的管道。

multi-echo-server/main.c

```
void setup_workers() {
    round_robin_counter = 0;

    // ...

    // launch same number of workers as number of CPUs
    uv_cpu_info_t *info;
    int cpu_count;
    uv_cpu_info(&info, &cpu_count);
    uv_free_cpu_info(info, cpu_count);

    child_worker_count = cpu_count;

    workers = calloc(sizeof(struct child_worker), cpu_count);
    while (cpu_count--) {
        struct child_worker *worker = &workers[cpu_count];
        uv_pipe_init(loop, &worker->pipe, 1);

        uv_stdio_container_t child_stdio[3];
        child_stdio[0].flags = UV_CREATE_PIPE | UV_READABLE_PIPE;
        child_stdio[0].data.stream = (uv_stream_t*) &worker->pipe;
        child_stdio[1].flags = UV_IGNORE;
        child_stdio[2].flags = UV_INHERIT_FD;
        child_stdio[2].data.fd = 2;

        worker->options.stdio = child_stdio;
        worker->options.stdio_count = 3;

        worker->options.exit_cb = close_process_handle;
        worker->options.file = args[0];
        worker->options.args = args;

        uv_spawn(loop, &worker->req, &worker->options);
        fprintf(stderr, "Started worker %d\n", worker->req.pid);
    }
}
```

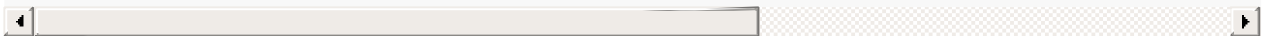
首先，我们使用酷炫的 `uv_cpu_info` 函数获取到当前的cpu的核心个数，所以我们也能启动一样数目的worker进程。再次强调一下，务必将 `uv_pipe_init` 的ipc参数设置为1。接下来，我们指定子进程的 `stdin` 是一个可读的管道（从子进程的角度来说）。接下来的一切就很直观了，worker进程被启动，等待着文件描述符被写入到他们的标准输入中。

在主进程的 `on_new_connection` 中，我们接收了client端的socket，然后把它传递给worker环中的下一个可用的worker进程。

multi-echo-server/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_write_t *write_req = (uv_write_t*) malloc(sizeof(uv_writ
        dummy_buf = uv_buf_init("a", 1);
        struct child_worker *worker = &workers[round_robin_counter]
        uv_write2(write_req, (uv_stream_t*) &worker->pipe, &dummy_b
        round_robin_counter = (round_robin_counter + 1) % child_wor
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```



`uv_write2` 能够在所有的情形上做了一个很好的抽象，我们只需要将client作为一个参数即可完成传输。现在，我们的多进程echo服务器已经可以运转起来啦。

感谢Kyle指出了 `uv_write2` 需要一个不为空的buffer。

Advanced event loops

libuv提供了非常多的控制event-loop的方法，你能通过使用多loop来实现很多有趣的功能。你还可以将libuv的event loop嵌入到其它基于event-loop的库中。比如，想象着一个基于Qt的UI，然后Qt的event-loop是由libuv驱动的，做着加强级的系统任务。

Stopping an event loop

`uv_stop()` 用来终止event loop。loop会停止的最早时间点是在下次循环的时候，或者稍晚些的时候。这也就意味着在本次循环中已经准备被处理的事件，依然会被处理，`uv_stop` 不会起到作用。当 `uv_stop` 被调用，在当前的循环中，loop不会被IO操作阻塞。上面这些说得有点玄乎，还是让我们看下 `uv_run()` 的代码：

src/unix/core.c - uv_run

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout;
  int r;
  int ran_pending;

  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);

  while (r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
    ran_pending = uv__run_pending(loop);
    uv__run_idle(loop);
    uv__run_prepare(loop);

    timeout = 0;
    if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEF
      timeout = uv_backend_timeout(loop);

    uv__io_poll(loop, timeout);
```

`stop_flag` 由 `uv_stop` 设置。现在所有的libuv回调函数都是在一次loop循环中被调用的，因此调用 `uv_stop` 并不能中止本次循环。首先，libuv会更新定时器，然后运行接下来的定时器，空转和准备回调，调用任何准备好的IO回调函数。如果你在它们之间的任何一个时间里，调用 `uv_stop()`，`stop_flag` 会被设置为1。

这会导致 `uv_backend_timeout()` 返回0，这也就是为什么loop不会阻塞在I／O上。从另外的角度来说，你在任何一个检查handler中调用 `uv_stop` ，此时I/O已经完成，所以也没有影响。

在已经得到结果，或是发生错误的时候， `uv_stop()` 可以用来关闭一个loop，而且不需要保证handler停止的顺序。

下面是一个简单的例子，它演示了loop的停止，以及当前的循环依旧在执行。

uvstop/main.c

```
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

void idle_cb(uv_idle_t *handle) {
    printf("Idle callback\n");
    counter++;

    if (counter >= 5) {
        uv_stop(uv_default_loop());
        printf("uv_stop() called\n");
    }
}

void prep_cb(uv_prepare_t *handle) {
    printf("Prep callback\n");
}

int main() {
    uv_idle_t idler;
    uv_prepare_t prep;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, idle_cb);

    uv_prepare_init(uv_default_loop(), &prep);
    uv_prepare_start(&prep, prep_cb);

    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    return 0;
}
```

Utilities

本章介绍的工具和技术对于常见的任务非常的实用。libuv吸收了libev用户手册页中所涵盖的一些模式，并在此基础上对API做了少许的改动。本章还包含了一些无需用完整的一章来介绍的libuv API。

Timers

在定时器启动后的特定时间后，定时器会调用回调函数。libuv的定时器还可以设定为，按时间间隔定时启动，而不是只启动一次。
可以简单地使用超时时间 `timeout` 作为参数初始化一个定时器，还有一个可选参数 `repeat` 。定时器能在任何时间被终止。

```
uv_timer_t timer_req;

uv_timer_init(loop, &timer_req);
uv_timer_start(&timer_req, callback, 5000, 2000);
```

上述操作会启动一个循环定时器（repeating timer），它会在调用 `uv_timer_start` 后，5秒（timeout）启动回调函数，然后每隔2秒（repeat）循环启动回调函数。你可以使用：

```
uv_timer_stop(&timer_req);
```

来停止定时器。这个函数也可以在回调函数中安全地使用。

循环的间隔也可以随时定义，使用：

```
uv_timer_set_repeat(uv_timer_t *timer, int64_t repeat);
```

它会在可能的时候发挥作用。如果上述函数是在定时器回调函数中调用的，这意味着：

> - 如果定时器未设置为循环，这意味着定时器已经停止。需要先用 `uv_timer_start` 重新启动。
> - 如果定时器被设置为循环，那么下一次超时的时间已经被规划好了，所以在切换到新的间隔之前，旧的间隔还会发挥一次作用。

函数：

```
int uv_timer_again(uv_timer_t *)
```

只适用于循环定时器，相当于停止定时器，然后把原先的 `timeout` 和 `repeat` 值都设置为之前的 `repeat` 值，启动定时器。如果当该函数调用时，定时器未启动，则调用失败（错误码为 `UV_EINVAL` ）并且返回－1。

下面的一节会出现使用定时器的例子。

Event loop reference count

event-loop在没有了活跃的handle之后，便会终止。整套系统的工作方式是：在handle增加时，event-loop的引用计数加1，在handle停止时，引用计数减少1。当然，libuv也允许手动地更改引用计数，通过使用：

```
void uv_ref(uv_handle_t*);
void uv_unref(uv_handle_t*);
```

这样，就可以达到允许loop即使在有正在活动的定时器时，仍然能够推出。或者是使用自定义的uv_handle_t对象来使得loop保持工作。

第二个函数可以和间隔循环定时器结合使用。你会有一个每隔x秒执行一次的垃圾回收器，或者是你的网络服务器会每隔一段时间向其他人发送一次心跳信号，但是你不想只有在所有垃圾回收完或者出现错误时才能停止他们。如果你想要在你其他的监视器都退出后，终止程序。这时你就可以立即unref定时器，即便定时器这时是loop上唯一还在运行的监视器，你依旧可以停止 `uv_run()` 。

它们同样会出现在node.js中，如js的API中封装的libuv方法。每一个js的对象产生一个 `uv_handle_t` （所有监视器的超类），同样可以被uv_ref和uv_unref。

ref-timer/main.c

```
uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

int main() {
    loop = uv_default_loop();

    uv_timer_init(loop, &gc_req);
    uv_unref((uv_handle_t*) &gc_req);

    uv_timer_start(&gc_req, gc, 0, 2000);

    // could actually be a TCP download or something
    uv_timer_init(loop, &fake_job_req);
    uv_timer_start(&fake_job_req, fake_job, 9000, 0);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

首先初始化垃圾回收器的定时器，然后在立刻 `unref` 它。注意观察9秒之后，此时fake_job完成，程序会自动退出，即使垃圾回收器还在运行。

Idler pattern

空转的回调函数会在每一次的event-loop循环激发一次。空转的回调函数可以用来执行一些优先级较低的活动。比如，你可以向开发者发送应用程序的每日性能表现情况，以便于分析，或者是使用用户应用cpu时间来做SETI运算:)。空转程序还可以用于GUI应用。比如你在使用event-loop来下载文件，如果tcp连接未中断而且当前并没有其他的事件，则你的event-loop会阻塞，这也就意味着你的下载进度条会停滞，用户会面对一个无响应的程序。面对这种情况，空转监视器可以保持UI可操作。

idle-compute/main.c

```
uv_loop_t *loop;
uv_fs_t stdin_watcher;
uv_idle_t idler;
char buffer[1024];

int main() {
    loop = uv_default_loop();

    uv_idle_init(loop, &idler);

    uv_buf_t buf = uv_buf_init(buffer, 1024);
    uv_fs_read(loop, &stdin_watcher, 0, &buf, 1, -1, on_type);
    uv_idle_start(&idler, crunch_away);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

上述程序中，我们将空转监视器和我们真正关心的事件排在一起。`crunch_away` 会被循环地调用，直到输入字符并回车。然后程序会被中断很短的时间，用来处理数据读取，然后在接着调用空转的回调函数。

idle-compute/main.c

```
void crunch_away(uv_idle_t* handle) {
    // Compute extra-terrestrial life
    // fold proteins
    // computer another digit of PI
    // or similar
    fprintf(stderr, "Computing PI...\n");
    // just to avoid overwhelming your terminal emulator
    uv_idle_stop(handle);
}
```

Passing data to worker thread

在使用 `uv_queue_work` 的时候，你通常需要给工作线程传递复杂的数据。解决方案是自定义struct，然后使用 `uv_work_t.data` 指向它。一个稍微的不同是必须让 `uv_work_t` 作为这个自定义struct的成员之一（把这叫做接力棒）。这么做就可以使得，同时回收数据和 `uv_wortk_t` 。

```
struct ftp_baton {
    uv_work_t req;
    char *host;
    int port;
    char *username;
    char *password;
}
```

```
ftp_baton *baton = (ftp_baton*) malloc(sizeof(ftp_baton));
baton->req.data = (void*) baton;
baton->host = strdup("my.webhost.com");
baton->port = 21;
// ...

uv_queue_work(loop, &baton->req, ftp_session, ftp_cleanup);
```

现在我们创建完了接力棒，并把它排入了队列中。

现在就可以随性所欲地获取自己想要的数据啦。

```
void ftp_session(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    fprintf(stderr, "Connecting to %s\n", baton->host);
}

void ftp_cleanup(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    free(baton->host);
    // ...
    free(baton);
}
```

我们既回收了接力棒，同时也回收了监视器。

External I/O with polling

通常在使用第三方库的时候，需要应对他们自己的IO，还有保持监视他们的socket和内部文件。在此情形下，不可能使用标准的IO流操作，但第三方库仍然能整合进event-loop中。所有这些需要的就是，第三方库就必须允许你访问它的底层文件描述符，并且提供可以处理有用户定义的细微任务的函数。但是一些第三库并不允许你这么做，他们只提供了一个标准的阻塞IO函数，此函数会完成所有的工作并返回。在event-loop的线程直接使用它们是不明智的，而是应该使用libuv的工作线程。当然，这也意味着失去了对第三方库的颗粒化控制。

libuv的 `uv_poll` 简单地监视了使用了操作系统的监控机制的文件描述符。从某方面说，libuv实现的所有的IO操作，的背后均有 `uv_poll` 的支持。无论操作系统何时监视到文件描述符的改变，libuv都会调用响应的回调函数。

现在我们简单地实现一个下载管理程序，它会通过libcurl来下载文件。我们不会直接控制libcurl，而是使用libuv的event-loop，通过非阻塞的异步的多重接口来处理下载，与此同时，libuv会监控IO的就绪状态。

uvwget/main.c - The setup

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>
#include <curl/curl.h>

uv_loop_t *loop;
CURLM *curl_handle;
uv_timer_t timeout;
}

int main(int argc, char **argv) {
    loop = uv_default_loop();

    if (argc <= 1)
        return 0;

    if (curl_global_init(CURL_GLOBAL_ALL)) {
        fprintf(stderr, "Could not init cURL\n");
        return 1;
    }

    uv_timer_init(loop, &timeout);

    curl_handle = curl_multi_init();
    curl_multi_setopt(curl_handle, CURLMOPT_SOCKETFUNCTION, handle_
    curl_multi_setopt(curl_handle, CURLMOPT_TIMERFUNCTION, start_t

    while (argc-- > 1) {
        add_download(argv[argc], argc);
    }

    uv_run(loop, UV_RUN_DEFAULT);
    curl_multi_cleanup(curl_handle);
    return 0;
}
```

每种库整合进libuv的方式都是不同的。以libcurl的例子来说，我们注册了两个回调函数。socket回调函数 `handle_socket` 会在socket状态改变的时候被触发，因此我们不得不开始轮询它。 `start_timeout` 是libcurl用来告知我们下一次的超时间隔的，之后我们就应该不管当前IO状态，驱动libcurl向前。这些也就是libcurl能处理错误或驱动下载进度向前的原因。

可以这么调用下载器：

```
$ ./uvwget [url1] [url2] ...
```

我们可以把url当成参数传入程序。

uvwget/main.c - Adding urls

```
void add_download(const char *url, int num) {
    char filename[50];
    sprintf(filename, "%d.download", num);
    FILE *file;

    file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening %s\n", filename);
        return;
    }

    CURL *handle = curl_easy_init();
    curl_easy_setopt(handle, CURLOPT_WRITEDATA, file);
    curl_easy_setopt(handle, CURLOPT_URL, url);
    curl_multi_add_handle(curl_handle, handle);
    fprintf(stderr, "Added download %s -> %s\n", url, filename);
}
```

我们允许libcurl直接向文件写入数据。

`start_timeout` 会被libcurl立即调用。它会启动一个libuv的定时器，使用 `CURL_SOCKET_TIMEOUT` 驱动 `curl_multi_socket_action` ，当其超时时，调用它。 `curl_multi_socket_action` 会驱动libcurl，也会在socket状态改变的时候被调用。但在我们深入讲解它之前，我们需要轮询监听socket，等待 `handle_socket` 被调用。

uvwget/main.c - Setting up polling

```
void start_timeout(CURLM *multi, long timeout_ms, void *userp) {
    if (timeout_ms <= 0)
        timeout_ms = 1; /* 0 means directly call socket_action, but
    uv_timer_start(&timeout, on_timeout, timeout_ms, 0);
}

int handle_socket(CURL *easy, curl_socket_t s, int action, void *us
    curl_context_t *curl_context;
    if (action == CURL_POLL_IN || action == CURL_POLL_OUT) {
        if (socketp) {
            curl_context = (curl_context_t*) socketp;
        }
        else {
            curl_context = create_curl_context(s);
            curl_multi_assign(curl_handle, s, (void *) curl_context
        }
    }

    switch (action) {
        case CURL_POLL_IN:
            uv_poll_start(&curl_context->poll_handle, UV_READABLE,
            break;
        case CURL_POLL_OUT:
            uv_poll_start(&curl_context->poll_handle, UV_WRITABLE,
            break;
        case CURL_POLL_REMOVE:
            if (socketp) {
                uv_poll_stop(&((curl_context_t*)socketp)->poll_hand
                destroy_curl_context((curl_context_t*) socketp);
                curl_multi_assign(curl_handle, s, NULL);
            }
            break;
        default:
            abort();
    }

    return 0;
}
```

我们关心的是socket的文件描述符s，还有action。对应每一个socket，我们都创造了 `uv_poll_t` ，并用 `curl_multi_assign` 把它们关联起来。每当回调函数被调用时， `socketp` 都会指向它。

在下载完成或失败后，libcurl需要移除poll。所以我们停止并回收了poll的handle。

我们使用 `UV_READABLE` 或 `UV_WRITABLE` 开始轮询，基于libcurl想要监视的事件。当socket已经准备好读或写后，libuv会调用轮询的回调函数。在相同的handle上调用多次 `uv_poll_start` 是被允许的，这么做可以更新事件的参数。 `curl_perform` 是整个程序的关键。

uvwget/main.c - Driving libcurl.

```
void curl_perform(uv_poll_t *req, int status, int events) {
    uv_timer_stop(&timeout);
    int running_handles;
    int flags = 0;
    if (status < 0)                      flags = CURL_CSELECT_ERR;
    if (!status && events & UV_READABLE) flags |= CURL_CSELECT_IN;
    if (!status && events & UV_WRITABLE) flags |= CURL_CSELECT_OUT;

    curl_context_t *context;

    context = (curl_context_t*)req;

    curl_multi_socket_action(curl_handle, context->sockfd, flags, &
    check_multi_info();
}
```

首先我们要做的是停止定时器，因为内部还有其他要做的事。接下来我们我们依据触发回调函数的事件，来设置flag。然后，我们使用上述socket和flag作为参数，来调用 `curl_multi_socket_action` 。在此刻libcurl会在内部完成所有的工作，然后尽快地返回事件驱动程序在主线程中急需的数据。libcurl会在自己的队列中将传输进度的消息排队。对于我们来说，我们只关心是否传输完成，这类消息。所以我们将这类消息提取出来，并将传输完成的handle回收。

uvwget/main.c - Reading transfer status.

```
void check_multi_info(void) {
    char *done_url;
    CURLMsg *message;
    int pending;

    while ((message = curl_multi_info_read(curl_handle, &pending)))
        switch (message->msg) {
        case CURLMSG_DONE:
            curl_easy_getinfo(message->easy_handle, CURLINFO_EFFEC
                            &done_url);
            printf("%s DONE\n", done_url);

            curl_multi_remove_handle(curl_handle, message->easy_han
            curl_easy_cleanup(message->easy_handle);
            break;

        default:
            fprintf(stderr, "CURLMSG default\n");
            abort();
        }
    }
}
```

Loading libraries

libuv提供了一个跨平台的API来加载共享库shared libraries。这就可以用来实现你自己的插件／扩展／模块系统，它们可以被nodejs通过 `require()` 调用。只要你的库输出的是正确的符号，用起来还是很简单的。在载入第三方库的时候，要注意错误和安全检查，否则你的程序就会表现出不可预测的行为。下面这个例子实现了一个简单的插件，它只是打印出了自己的名字。

首先看下提供给插件作者的接口。

plugin/plugin.h

```
#ifndef UVBOOK_PLUGIN_SYSTEM
#define UVBOOK_PLUGIN_SYSTEM

// Plugin authors should use this to register their plugins with mf
void mfp_register(const char *name);

#endif
```

你可以在你的程序中给插件添加更多有用的功能（mfp is My Fancy Plugin）。使用了这个api的插件的例子：

plugin/hello.c

```
#include "plugin.h"

void initialize() {
    mfp_register("Hello World!");
}
```

我们的接口定义了，所有的插件都应该有一个能被程序调用的 `initialize` 函数。这个插件被编译成了共享库，因此可以被我们的程序在运行的时候载入。

```
$ ./plugin libhello.dylib
Loading libhello.dylib
Registered plugin "Hello World!"
```

Note

> 共享库的后缀名在不同平台上是不一样的。在Linux上是libhello.so。

使用 `uv_dlopen` 首先载入了共享库 `libhello.dylib` 。再使用 `uv_dlsym` 获取了该插件的 `initialize` 函数，最后在调用它。

plugin/main.c

```
#include "plugin.h"

typedef void (*init_plugin_function)();

void mfp_register(const char *name) {
    fprintf(stderr, "Registered plugin \"%s\"\n", name);
}

int main(int argc, char **argv) {
    if (argc == 1) {
        fprintf(stderr, "Usage: %s [plugin1] [plugin2] ...\n", argv
        return 0;
    }

    uv_lib_t *lib = (uv_lib_t*) malloc(sizeof(uv_lib_t));
    while (--argc) {
        fprintf(stderr, "Loading %s\n", argv[argc]);
        if (uv_dlopen(argv[argc], lib)) {
            fprintf(stderr, "Error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin_function init_plugin;
        if (uv_dlsym(lib, "initialize", (void **) &init_plugin)) {
            fprintf(stderr, "dlsym error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin();
    }

    return 0;
}
```

函数 `uv_dlopen` 需要传入一个共享库的路径作为参数。当它成功时返回0，出错时返回－1。使用 `uv_dlerror` 可以获取出错的消息。

`uv_dlsym` 的第三个参数保存了一个指向第二个参数所保存的函数的指针。 `init_plugin_function` 是一个函数的指针，它指向了我们所需要的程序插件的函数。

TTY

文字终端长期支持非常标准化的控制序列。它经常被用来增强终端输出的可读性。例如 `grep --colour` 。libuv提供了跨平台的， `uv_tty_t` 抽象（stream）和相关的处理ANSI escape codes 的函数。这也就是说，libuv同样在Windows上实现了对等的ANSI codes，并且提供了获取终端信息的函数。

首先要做的是，使用读／写文件描述符来初始化`uv_tty_t`。如下：

```
int uv_tty_init(uv_loop_t*, uv_tty_t*, uv_file fd, int readable)
```

设置`readable`为true，意味着你打算使用`uv_read_start`从stream从中读取数据。

最好还要使用`uv_tty_set_mode`来设置其为正常模式。也就是运行大多数的TTY格式，流控制和其他的设置。其他的模式还有这些。

记得当你的程序退出后，要使用`uv_tty_reset_mode`恢复终端的状态。这才是礼貌的做法。另外要注意礼貌的地方是关心重定向。如果使用者将你的命令的输出重定向到文件，控制序列不应该被重写，因为这会阻碍可读性和grep。为了保证文件描述符确实是TTY，可以使用`uv_guess_handle`函数，比较返回值是否为`UV_TTY`。

下面是一个把白字打印到红色背景上的例子。

tty/main.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t *loop;
uv_tty_t tty;
int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, 1, 0);
    uv_tty_set_mode(&tty, UV_TTY_MODE_NORMAL);

    if (uv_guess_handle(1) == UV_TTY) {
        uv_write_t req;
        uv_buf_t buf;
        buf.base = "\033[41;37m";
        buf.len = strlen(buf.base);
        uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    }

    uv_write_t req;
    uv_buf_t buf;
    buf.base = "Hello TTY\n";
    buf.len = strlen(buf.base);
    uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    uv_tty_reset_mode();
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

最后要说的是`uv_tty_get_winsize()`，它能获取到终端的宽和长，当成功获取后返回0。下面这个小程序实现了一个动画的效果。

tty-gravity/main.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t *loop;
uv_tty_t tty;
uv_timer_t tick;
uv_write_t write_req;
int width, height;
int pos = 0;
char *message = "  Hello TTY  ";

void update(uv_timer_t *req) {
    char data[500];

    uv_buf_t buf;
    buf.base = data;
    buf.len = sprintf(data, "\033[2J\033[H\033[%dB\033[%luC\033[42
                            pos,
                            (unsigned long) (width-strlen(message))
                            message);
    uv_write(&write_req, (uv_stream_t*) &tty, &buf, 1, NULL);

    pos++;
    if (pos > height) {
        uv_tty_reset_mode();
        uv_timer_stop(&tick);
    }
}

int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, 1, 0);
    uv_tty_set_mode(&tty, 0);

    if (uv_tty_get_winsize(&tty, &width, &height)) {
        fprintf(stderr, "Could not get TTY information\n");
        uv_tty_reset_mode();
        return 1;
    }

    fprintf(stderr, "Width %d, height %d\n", width, height);
    uv_timer_init(loop, &tick);
    uv_timer_start(&tick, update, 200, 200);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

escape codes的对应表如下：

代码	意义
2 J	Clear part of the screen, 2 is entire screen
H	Moves cursor to certain position, default top-left
n B	Moves cursor down by n lines
n C	Moves cursor right by n columns
m	Obeys string of display settings, in this case green background (40+2), white text (30+7)

正如你所见，它能输出酷炫的效果，你甚至可以发挥想象，用它来制作电子游戏。更有趣的输出，可以使用 `http://www.gnu.org/software/ncurses/ncurses.html` 。

About

Nikhil Marathe在某一个下午(June 16, 2012)开始写这本书。当他在写node-taglib的时候苦于没有好的libuv文档。虽然已经有了官方文档，但是没有好理解的教程。本书正是应需求而生，并且努力变得准确。也就是说，本书中可能会有错误。所以鼓励大家Pull requests。你当然可以直接给他发email，告诉他错误。

Nikhil从Marc Lehmann的关于libev的手册中讲解libev和libuv的不同点的部分，获取了不少的灵感。

本书的中文翻译者为：luohaha，byronhe，littleneko。同样欢迎您的Pull requests来改进本书的翻译工作。

Licensing